AutoToken: Predicting Peak Parallelism for Big Data Analytics at Microsoft

Rathijit Sen Alekh Jindal Hiren Patel Shi Qiao
{rathijit.sen, alekh.jindal, hirenp, shqiao}@microsoft.com

Microsoft Corporation

ABSTRACT

Right-sizing resource allocation for big-data queries, particularly in serverless environments, is critical for improving infrastructure operational efficiency, capacity availability, query performance predictability, and for reducing unnecessary wait times. In this paper, we present AutoToken — a simple and effective predictor for estimating the peak resource usage of recurring big data queries. It uses multiple query plan identifiers to identify recurring query templates and to learn models with the goal of reducing over-allocation in future instances of those queries. AutoToken is computationally light, for both training and scoring, is easily deployable at scale, and is integrated with the Peregrine workload optimization infrastructure at Microsoft. We extensively evaluate AutoToken on SCOPE jobs from our production clusters and show that it outperforms state-of-the-art solutions for peak resource estimation.

We also discuss our plans towards supporting repeatable and extensible research on resource prediction for SCOPE jobs, including describing a simulation methodology for generating arbitrary-sized datasets with similar characteristics as the production datasets.

PVLDB Reference Format:
Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. AutoToken: Predicting Peak Parallelism for Big Data Analytics at Microsoft. *PVLDB*, 13(12): 3326-3339, 2020.
DOI: https://doi.org/10.14778/3415478.3415554

1. INTRODUCTION

We have witnessed a decade of tremendous interest in large scale data processing, and consequently the rise of so called big data systems. While the early focus was on handling the scale and complexity of big data, it is increasingly critical to improve the resource efficiency and reduce operational costs in massive data processing infrastructures. As a result, efficient resource allocation at the data center level has received a lot of attention in recent times [15, 37, 23, 31, 7, 12, 13, 5]. Interestingly, resource efficiency becomes harder with the new breed of so called *serverless* query processing, where users do not have to setup clusters. Instead, the cloud provider takes care of allocating resources on a per-query basis, e.g., Athena [1], Big Query [16], SCOPE [11], etc. While serverless query processing makes it easier for the users to start processing massive datasets, it is challenging for the cloud providers to estimate the resource requirements at the query level. This is because the relationship between the resources provided and the performance observed for a query is often non-intuitive and even domain experts would struggle to manually pick the right set of resources for a given query [31]. Thus, efficient use of resources in modern big data infrastructures is a challenge for cloud providers and other large enterprises.

In this paper, we analyze the resource allocation effectiveness in the Cosmos big data analytics infrastructure at Microsoft. The bulk of the workload in this infrastructure consists of batch processing using the SCOPE query engine, which is exposed as a *job service* to the users, i.e., users submit their declarative SCOPE scripts and the engine takes care of picking the resources (e.g., the number of containers) for processing that script. Multiple prior works have considered the problem of efficient resource allocation in SCOPE [15, 23, 31]. Most recently, Morpheus [23] uses historical data to produce the resource allocation skyline of *recurring jobs*, i.e., SCOPE scripts that are executed periodically with different inputs and parameters. Yet, as we show in this paper, the resource allocation efficiency in SCOPE still has major gaps, with around 40–60% of the jobs over-allocated in different clusters, by as much as 1000×. The reason Morpheus is hard to apply here is because it considers a SCOPE job as a black box, i.e., it monitors resource consumption from the YARN logs without considering the SCOPE query plans, and applies a single resource allocation skyline to all instances of a recurring job. In practice, however, recurring jobs instances have several variations, the most notable being changes in inputs sizes which can easily grow or shrink by 2× in different instances.

Thus, we argue for building *machine learning models* with a richer set of features that can capture the resource allocation requirements of different instances of a recurring job more precisely. In fact, our results show that such a model could be two orders of magnitude more accurate than Morpheus. We use this learned resource model to right-size the peak allocation of over-allocated jobs, since majority of the SCOPE jobs are over-allocated anyways. Training this model requires grouping instances of a recurring job and we consider a rigid and a relaxed (to allow for more variations) grouping strategies, which expose the trade-off between accuracy and coverage over the workload. We also explore other design choices, such as model type, training support, and training interval, that are important factors when deploying machine learning models to production. We explore these design choices with our production settings in mind and present an extensive evaluation over a large workload of 8.8 million production SCOPE jobs collected over a period of five weeks. We show the end-to-end inte-

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415554

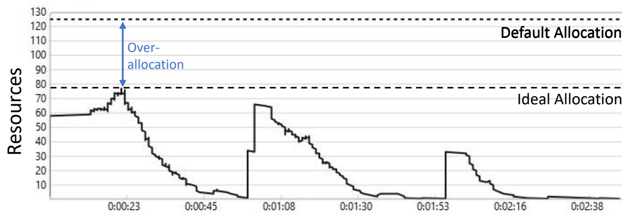


Figure 1: Resource usage in a typical SCOPE job over time.

gration of the resulting system, AutoToken, with the SCOPE query engine. Finally, we describe the repeatability of our experiments using a simulator to mimic our production workloads.

In summary, our key contributions are as follows:

- We present an overview of the Cosmos big data infrastructure at Microsoft, the current strategy for peak resource allocation, an evaluation of the effectiveness of this strategy, and the potential capacity and queuing time savings. (Section 2)
- We describe our production scenario consisting of recurring jobs, the workload trace of 8.8 million jobs from over five weeks, and the production requirements for shipping a new strategy for peak resource allocation. (Section 3)
- We present AutoToken for better peak allocation accuracy using machine learning models, that have 50th percentile error of close to 0 and 90th percentile error of 50% or less, orders of magnitude lower than the 50th and 90th percentile errors of up to 4900% and 24500% with the current default. (Section 4)
- We further evaluate and discuss various design choices in AutoToken over a filtered set of *production candidates*, constituting around 12–29% of our entire production workloads, in order to meet our requirements. (Section 5)
- We compare AutoToken with both a conservative allocation strategy and with Morpheus, the prior art for optimizing the peak resources, over the production candidates. AutoToken has up to two orders to magnitude lower RMSE than both the conservative strategy and Morpheus. (Section 6)
- We discuss the production readiness of AutoToken, its integration with the SCOPE query engine, and show pre-production experiments on customer workloads. (Section 7)
- Finally, we describe the repeatability of our experiments using a dataset simulation tool that mimics our production workload traces. (Section 8)

2. SCOPE RESOURCE ALLOCATION: IMPERFECTIONS & OPPORTUNITIES

We now present an overview of Cosmos big data analytics infrastructure at Microsoft, with SCOPE as the primary query engine. A typical SCOPE job consists of stages that are connected in a directed acyclic graph (DAG). Each stage further contains one or more physical operators that could be processed locally in a single container, and instances of a stage (called vertices) can process different partitions of data in parallel. For the sake of presentation, we limit our discussion to maximum degree of parallelism (also referred to as resource *tokens* in SCOPE) as the unit of resource, however, one could easily extend it to other dimensions such as container size, VM type, etc.

SCOPE and other modern big data systems, including serverless query processing systems, shifts the onus of resource allocation from users to the query engines. This is challenging due to several reasons. First of all, it is hard to estimate the fine-grained resource requirements for each query at compile time. The estimates in a query optimizer are often off by orders of magnitudes

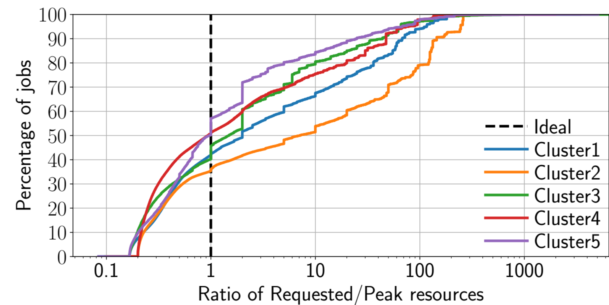


Figure 2: Distributions of the percentage of jobs as a function of of (Requested/Peak) resource allocations (x-axis, log scale).

and the problem gets worse in big data systems. Second, allocating resources is expensive and so it is not desirable to change the allocation of a query frequently. Furthermore, a resource change can trigger query plan changes to use the new set of resources, which can make the overall performance worse. And finally, given that a query could consume maximum resources early on, there may not be much room for adjusting the resources, particularly if the resources are under allocated then query performance may have suffered already before any adjustments could be applied.

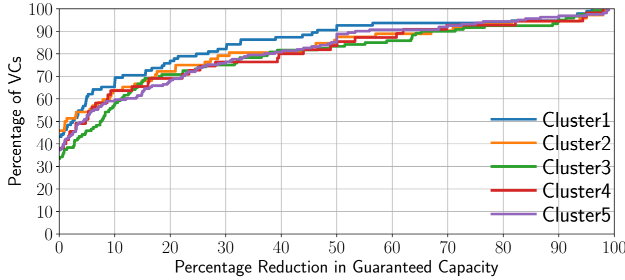
SCOPE addresses some of the above issues by relying on a user-specified resource limit, i.e., the maximum number of tokens that a query could use, and reserves them as guaranteed resources before starting the query execution. We call this the *Default allocation*. Unfortunately, users rarely make an informed decision when specifying the maximum tokens. Figure 1 shows the token usage skyline in a typical SCOPE job. We can see the big gap between the Default allocation and the actual peak token consumption (Ideal allocation). Over-provisioning of tokens results in resource wastage, unnecessary waits, and can reduce cluster utilization.

On the other hand, for under-allocation, SCOPE tries to opportunistically use spare tokens [9]. However, spare tokens are not guaranteed and can still result in under-allocation with an accompanying loss in performance and/or unpredictability in query performance. Interestingly, some users *choose to under-allocate* jobs while prioritizing cost-savings over predictably-good performance. Thus, fixing under-allocations can be tricky, since we need to consider the user intent. We do not address this scenario in this work.

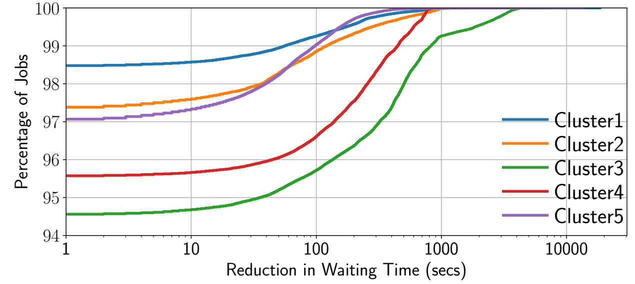
2.1 Allocation Effectiveness

As mentioned above, SCOPE jobs rely on a user provided maximum number of tokens to reserve resources before the job starts executing. Therefore, we first analyze the effectiveness of this resource allocation. Figure 2 shows the (cumulative) distribution of the percentage of jobs as a function of the *allocation ratio* — the ratio of the requested (guaranteed) to the actual peak consumed resources — of SCOPE jobs in five different production clusters at Microsoft. Allocation ratios of >, =, and < 1 represent over-, exact-, and under-allocations respectively. The vertical dashed line represents the *Ideal* scenario (allocation ratio = 1). Interestingly, we see from Figure 2 that around 40–60% of the jobs are over-allocated in different clusters, by as much as 1000×, indicating significant opportunities for right-sizing the resource allocation. The distribution for Cluster5 is closest to the Ideal since the business unit on that cluster spent considerable effort to build client-side tools for resource allocation. Despite this, Cluster5 still has 40% of the jobs over-allocated, with 15% over-allocated more than 10×.

Reducing the above identified over-allocation has several implications. First of all, it improves the operational efficiency in



(a) Guaranteed Capacity reduction



(b) Absolute wait time reduction

Figure 3: Distributions of savings opportunities if no jobs are over-allocated, for (a) guaranteed VC capacities but with no change in job wait times and (b) job wait times but with no change in VC capacities. The results are for the DT datasets (see Section 3.2) of each cluster.

Algorithm 1: Estimating min. guaranteed VC capacity

```
   Input: VC, List of jobs run on that VC
   Output: VC_Capacity
1  capacity = 0, ctr = 0
2  heap = {}
3  for each job j in list do
4      insert tuple [j, j.start_offset, JOB_START] in heap
5  while heap not empty do
6      extract_min tuple=[j, offset, mode] from heap
7      job_req=tokens(j)
8      if mode == JOB_START then
9          ctr = ctr - job_req
10         if ctr < capacity then
11             capacity = ctr
12         insert tuple [j, j.end_offset, JOB_END] in heap
13     else
14         ctr = ctr + job_req
15 VC_Capacity = -capacity
```

a highly valuable business that powers big data analytics across the whole of Microsoft, including products such as Office, Windows, Bing, Xbox, etc. Second, it frees up guaranteed resources that could be used to submit more SCOPE jobs. Third, it reduces the queuing time of jobs by having them request for less resources. Finally, it improves the user experience by automating a mandatory parameter in SCOPE jobs. We quantify some of the benefits below.

2.2 Potential Capacity Savings

SCOPE clusters are logically partitioned into *Virtual Clusters* (VCs), each mapping roughly to a business function or a business unit and with an exclusive capacity of guaranteed tokens that can be allocated to jobs submitted to that VC. To quantify the opportunity for capacity reductions, we performed an offline simulation of our jobs to infer the minimum number of tokens in each VC so that all the jobs that actually ran on that VC can have their guaranteed tokens requirements be satisfied.

Algorithm 1 shows the main steps of the simulation. The high-level idea is to use event-driven simulation to approximate the behavior of the SCOPE job scheduler, by keeping track of capacity reservations when jobs are submitted and capacity release when they finish. The simulator maintains a per-VC min-heap to use as a priority queue for scheduling job start and end events. It uses the actual job start and end times, calculated as offsets from a fixed timestamp, to compare across elements for maintaining the heap property. Lines 10 – 11, 15 keep track of the largest capacity requirement. This is the minimum capacity that the VC should have if job latencies are not to be affected. A lower VC capacity would result in jobs waiting longer for the required tokens to be available.

We ran Algorithm 1 twice, once with the Default token allocations, and once assuming that no job is over-allocated. Figure 3a shows the distribution of the percentage of VCs as a function of the percentage in reduction of minimum capacities. Across the clusters, ~60% of VCs show reduced capacity needs, with ~7–16% of VCs show a capacity reduction of 50% or more if the over-allocation problem can be eliminated. The capacity requirements of the entire cluster is the sum of the capacity requirements of its constituent VCs. The cluster-level capacity reductions are also significant — Cluster1: 10.6%, Cluster2: 9.1%, Cluster3: 13.6%, Cluster4: 11.4%, Cluster5: 12.3%. Reduced capacity requirements result in COGS (cost of goods and services) savings while supporting the same workload, or being able to accept more jobs (higher job concurrency) for the same capacity, or both.

2.3 Potential Queueing Improvements

To quantify the opportunity for job waiting time reductions, we performed another offline simulation using the estimated minimum VC capacities for default job allocations. Algorithm 2 shows the main steps of the event-driven simulation. It is similar to the capacity simulation, but now we allow a job to be scheduled within the time interval [submit_offset + min_delay, start_offset] where min_delay refers to the minimum delay between when a job is submitted to when it can be started owing to compilation and other setup requirements. As a simplification for this analysis, we calculate min_delay as a constant from the actual time offsets in our datasets (line 4). In practice, the delay would depend on job characteristics that affect, for example, job compilation time. While the jobs can be scheduled earlier than their actual start times in the simulation, we constrain the job starting order to be the same as in the actual dataset. For this reason we maintain the job_order_list (line 3) and start jobs in the sequence in which they appear in that list (lines 15–23). In contrast with the capacity simulation, we also need to keep track of simulated time that we advance whenever an element is extracted from the heap.

Algorithm 2 is a simplified approximation of the SCOPE job scheduler that takes into account more factors such as job priorities, job-dependent compilation and setup times, etc. Nevertheless, our estimated wait times with default token allocations and estimated VC capacities (from Algorithm 1) show good correlation with observed wait times for the jobs. The correlation coefficients between simulated and actual waits were as follows — Cluster1:

Algorithm 2: Estimating job wait times

```
   Input: VC, List of jobs run on that VC, VC_capacity
   Output: wait_times
1  cur_offset = 0
2  ready_list = [], heap = {}
3  job_order_list = list of jobs sorted by start_offset
4  min_delay = ∀_j min(j.start_offset - j.submit_offset)
5  for each job j in list do
6  |   insert tuple [j, j.submit_offset + min_delay,
   |     JOB_READY] in heap
7  while heap not empty do
8  |   dequeue tuple=[j, offset, mode] from heap
9  |   cur_offset=offset
10 |   job_req=tokens(j)
11 |   if mode == JOB_READY then
12 |   |   enqueue j in ready_list
13 |   else
14 |   |   capacity = capacity + job_req
15 |   repeat
16 |   |   k = next job in job_order_list
17 |   |   req = tokens(k)
18 |   |   if k in ready_list and req < capacity then
19 |   |   |   capacity = capacity - req
20 |   |   |   insert tuple [k, cur_offset + k.run_time,
   |   |   |     JOB_END] in heap
21 |   |   |   wait_times[k] = cur_offset - k.submit_offset
22 |   |   |   dequeue k from ready_list
23 |   until no more jobs dequeued;
```

0.95, Cluster2: 0.99, Cluster3: 0.97, Cluster4: 0.98, Cluster5: 0.9.

We ran Algorithm 2 twice, once with the default token allocations, and once assuming that no job is over-allocated but the VC capacities unchanged. Figure 3b show the distribution of the percentage of jobs as a function of the reduction of wait times. A reduction of close to 100% in wait time means that the job could start immediately when ready, under no over-allocations for any job, instead of waiting for tokens, under Default token allocations for jobs. While the percentage of jobs having reduced waits is not very high (up to 4–5%), the magnitude of reductions can be large, from several minutes to hours. Reduced wait times not only improve customer experience, they also enable higher cluster utilization as more jobs may be concurrently started due to the reduced token requirements.

3. PRODUCTION SCENARIO

In this section, we describe our production scenario, including the type of workload that we are going to focus on, the production trace that we are going to use throughout the paper, and the production requirements that drive many of our decisions.

3.1 Recurring Workloads

As we discussed before, we focus on recurring jobs in this work since they are business critical and form the majority of production workloads. Recurring jobs are periodic jobs that have the same query template getting executed with different inputs and parameters each time. For example, an hourly job analyzing customer engagement in the last 24 hours will execute with sliding filter predicates (and hence newer inputs) every hour. Recurring jobs could be hourly, daily, weekly, monthly, or something in between. To illustrate, Figure 4 shows 152 instances of an hourly recurring job, over six and a half day period, that extracts facts from a production clickstream. Over these 152 instances of this recurring, the execution time varies from 40 minutes to 2 hours and 25 minutes, while the input size varies from 65,290 GB to 165,269 GB. Therefore, we need to consider these variations in recurring job characteristics for resource prediction.

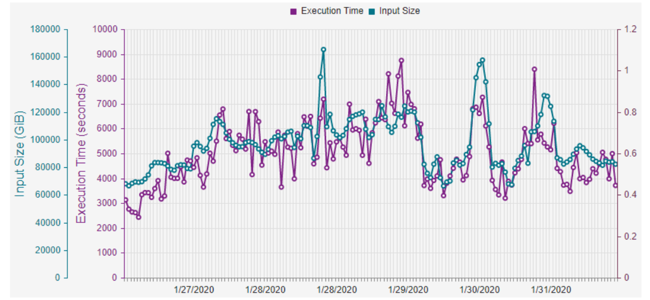


Figure 4: An hourly SCOPE *recurring job*.

Table 1: Characteristics of datasets used in this study.

(a) Number of jobs

Dataset	Cluster1	Cluster2	Cluster3	Cluster4	Cluster5
DR (Training)	1903095	793288	1073245	1112767	2374582
DT (Testing)	404208	161703	195449	238182	528481

(b) Number of VCs

Dataset	Cluster1	Cluster2	Cluster3	Cluster4	Cluster5
DR (Training)	165	100	204	106	171
DT (Testing)	95	72	120	55	161

Identifying different instances of a recurring job is tricky. A naive way could be to use the job name and normalize any instance specific details, e.g., the dates or ids, similar to as the one used in Morpheus [23]. Another approach could be to hash the text of the SCOPE scripts. However, either approach risks grouping dissimilar jobs together. For example, a generic name such as `Scope.script` may be used for many very different SCOPE scripts. Therefore, we group different instances of a recurring job using a *recurring signature*, a hash that uniquely identifies a recurring job using the query plan characteristics instead of using job name or script text. We hash the optimized logical query plan while normalizing the inputs and the parameter values. Note that prior works have also used such signatures for other applications such as computation reuse [22, 32], learning cardinalities [39], and learning cost models [34].

Our strategy is to group jobs that have the same recurring signature and to train a token prediction model for each group. We can then use the model to predict token requirements for future jobs that have the same recurring signature as that for the group. For any group g, $n(g)$ denotes the number of jobs belonging in that group. In order to train models with good accuracy, we require $n(g)$ to be above a certain threshold value. We call this threshold the support, S. We will train models only for groups that have $n(g) \geq S$.

3.2 Production Trace

We gathered the production trace from past job runs on our production clusters and anonymized values to remove any identifiable and sensitive information in the workloads. Table 1 describes the aggregate characteristics of our datasets from five of our production clusters. For each cluster, we have a Training dataset, that corresponds to a subset of jobs submitted to that cluster over a month,

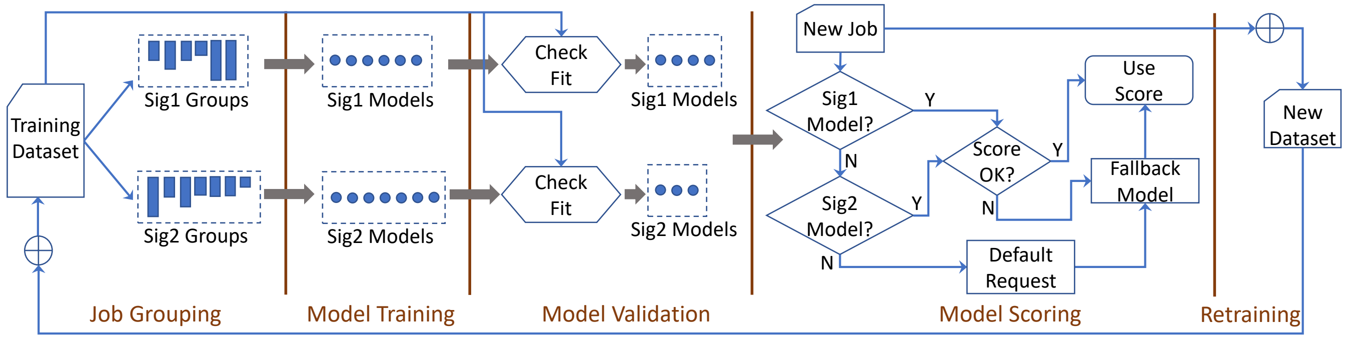


Figure 5: Overview of AutoToken Operations.

and a Testing dataset that corresponds to a subset of jobs submitted in the following week. For a given cluster c, we refer to these datasets as DR(c) and DT(c) respectively, or simply as DR and DT when c is clear from the context. We train models on jobs in DR and predict token counts for jobs in DT.

For each dataset we describe the number of jobs (Table 2a) and the number of VCs (Table 2b). For each cluster, since DR covers a longer time interval than DT, the counts are larger than those for DT. There is considerable variation in the counts across clusters. Cluster5 has the largest number of jobs (~2.37M in DR, ~0.53M in DT) while Cluster2 has the smallest (~0.79M in DR, ~0.16M in DT). Cluster3 has the largest number of VCs. The average number of jobs per VC varies from ~1.6K in DT(Cluster3) to ~13.9K in DR(Cluster5). Despite these variations, as we shall show, AutoToken works well for all of these clusters.

3.3 Requirements

We now discuss the key requirements when improving the parallelism in SCOPE jobs, as derived from our production workloads.

R1. The resource allocation predictions must be highly accurate, and consistently so over a time period of time.

R2. The prediction models must be applicable to a substantial portion of the workload in order to realize the potential gains.

R3. All training steps and their tuning must be automatic and must happen offline completely agnostic to the users.

R4. We want to minimize any performance regressions that affect the customers due to changes in SCOPE job parallelism.

R5. Adjusting job parallelism should not stress the clusters with additional resource requirements.

R6. Resource models should adapt with recurring job fluctuations in the workload over time.

R7. We want to leverage existing workload optimization infrastructure [21] for offline training and feedback.

R8. And finally, we need to provide compiler flags for users to control whether the feature is turned on or not.

4. AUTOTOKEN

In the previous section, we saw how the job characteristics in recurring workloads vary over time. Therefore, we want to consider a feature-based learning approach that can potentially lead to much better accuracy. In this section, we describe the design of the AutoToken predictor that we use to predict the peak resource usage of recurring jobs. Figure 5 illustrates the five main steps in AutoToken, which we discuss below.

Job Grouping: The brute force approach could be to feed all the query plans and metrics into a machine learning pipeline, and learn a single global model to predict the peak resource usage in future jobs. However, prior works have shown that it is extremely hard to get such a single global right [39, 34]. Indeed, we tried building a global model and the accuracy was poor (see Section 5.4). The reason global models are hard to train is due to the combinatorial space of all possible query plans. As a result, even with millions of queries in the cloud workloads, it is really hard to capture the resource behavior of query plans in a single monolithic model. Therefore, similar to the prior works, we consider building a large number of specialized machine learning models that are expected to be very accurate (**R1**). Specifically, we first group jobs based on their recurring signatures and then learn one model per template.

Model Training: The resources required for a job depend on job characteristics such as input size and job parameters. Therefore, a regression analysis over these variables from the historical query workload is expected to yield good predictions, i.e., we can train models offline and agnostic to the users (**R3**). Thus, we train resource models for each recurring job group provided, with a minimum support requirement (S) to ensure consistency over a period of time (**R1**). We start with S = 10 and Linear Regression models with inputs not standardized (LR_NS), but later we evaluate different choices in Section 5. We use off-the-shelf implementations from scikit-learn [30] for all model training in this work.

Model Validation: After training the models, we perform a validation step where we use the models to predict (fit) for jobs in the training dataset. The goal is to filter out models that might lead to performance regressions (**R4**). Any model that results in the predicted resource usage for any job, belonging to that group, being less than the actual peak by a threshold, e.g., 10 tokens, is discarded. This reduces prediction coverage but usually reduces the worst-case predictions significantly.

Model Scoring: When a new job arrives, we compute its signature and check if a resource model exists for that signature. AutoToken could use multiple signatures to group jobs in different ways and cover large portions of the workload (**R2**). For instance, Figure 5 shows two signatures that are computed for every job. If a model exists, we use it to predict the token count for the job. Otherwise, we use the Default value. For evaluating AutoToken, we use the Testing datasets (DT) (see Section 3.2) for model scoring.

Despite the model validation step, we cannot provide worst-case limits on the over- or under-allocations that may result from using the values predicted by AutoToken. To protect against the rare but large errors (**R4**), we use the predicted values only if they lie within a predetermined range, e.g., [1, 2× the default value]. If it lies outside this range, we use a fallback model, e.g., average of the actual peak token counts for jobs belonging to that group.

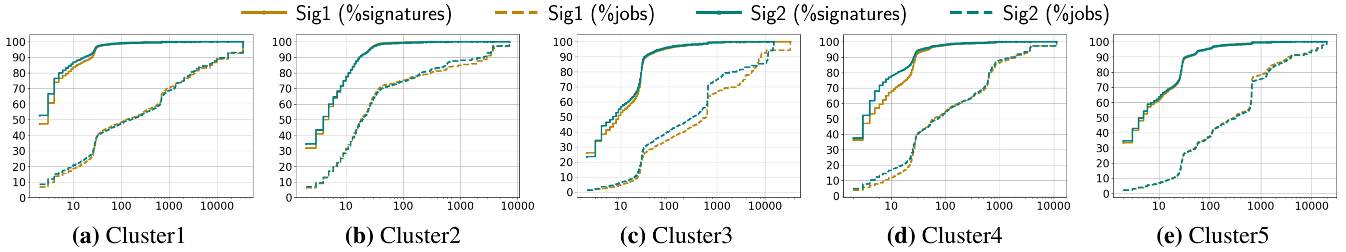


Figure 6: Distributions of the percentage of groups (y-axis, solid lines) and percentage of jobs belonging to those groups (y-axis, dashed lines) as a function of $n(g)$ (x-axis, log scale) for the DR datasets (see Section 3.2) considering only groups that have $n(g) \geq 2$ and no under-allocated job. Table 3 describes the counts of signatures and jobs corresponding to values of 100% in this figure.

Retraining: Once the newly arrived jobs have executed, we know their actual peak resource usages. We then add these jobs to the training dataset, retrain and validate models for groups where the jobs got added. In this way we keep up with changing workload characteristics over time (**R6**). Section 5.3 discusses the impact of the granularity of this retraining interval .

In the following, first we describe different granularities for defining job templates, and then we discuss the accuracy and coverage of our predictions.

4.1 Identifying Job Templates

The first question in AutoToken is to define templates that are general enough to cover a wide workload, and yet sufficiently specialized to learn accurate models. Unfortunately, it is hard to achieve both using a single signature [34], therefore AutoToken supports using multiple signatures. Specifically, we consider the following two signatures to identify recurring jobs: (1) a primary signature ($Sig1$) that recursively computes the numerical hash of each operator in the logical query plan, while excluding the parameters and inputs since they can change over time, and (2) a secondary signature ($Sig2$) that hashes just the inputs and outputs of the query plan, while allowing the plan operators and graph structure to change in between. The motivation for using the secondary signature is to increase prediction coverage — it may be possible to predict its peak resource usage of jobs consuming the same inputs even though their query plans may look very different.

Table 2: Number of distinct job signatures and distinct non-recurring (NR) job signatures as a percentage of total job signatures in the DR datasets.

	Cluster1	Cluster2	Cluster3	Cluster4	Cluster5
#Sig1	57.6%	57.4%	17.9%	24.1%	19.1%
#Sig2	53.0%	54.9%	16.5%	24.1%	17.6%
#Sig1_NR	53.2%	52.7%	14.4%	19.2%	15.8%
#Sig2_NR	47.8%	49.7%	13.0%	18.5%	14.4%

While describing the groupings for our training datasets, DR, we use the total number of jobs in DR (from Table 1) as a reference for calculating percentages. Table 2 shows the numbers (as percentages) of distinct and distinct non-recurring (NR) signatures. For example, in Cluster1, these numbers for $Sig1$ are 57.6% and 53.2% respectively of the total number of jobs in DR(Cluster1). Thus, the number of recurring signatures (with more than one job belonging to that group) is 57.6% − 53.2% = 4.4% of the total number of jobs in DR(Cluster1). Clusters 3 – 5 have a relatively smaller percentage of non-recurring signatures compared to clusters 1 and 2. Across all clusters, the number of a small number of recurring signatures (< 5%) cover all large fraction of total jobs. This makes it feasible to train and deploy per-signature models for recurring signatures.

4.2 Production Candidates

Our approach in this work is to train per-signature models that learn peak token counts from past job executions (recurring jobs). We do not target under-allocations, but want to predict accurately for over-allocated jobs and (the small fraction of) jobs that are already correctly allocated, i.e., jobs with allocation ratio ≥ 1. While Figure 2 shows ~40–60% of jobs have allocation ratios of ≥ 1, a fraction of those correspond to non-recurring/ad-hoc jobs which would not be covered by our approach. Therefore, the question is what maximum prediction coverage can be achieved if we only consider recurring jobs with allocation ratio ≥ 1 in DT such that the corresponding $Sig1$ or $Sig2$ also appear in DR and have a minimum S in DR. Figure 10a shows that such a coverage ranges from 14.3% (S = 1, Cluster2) to 38% (S = 1, Cluster5). S = 2 and S = 10 have slightly less coverage, but S = 100 reduces this significantly, to 9.2% – 28.4%.

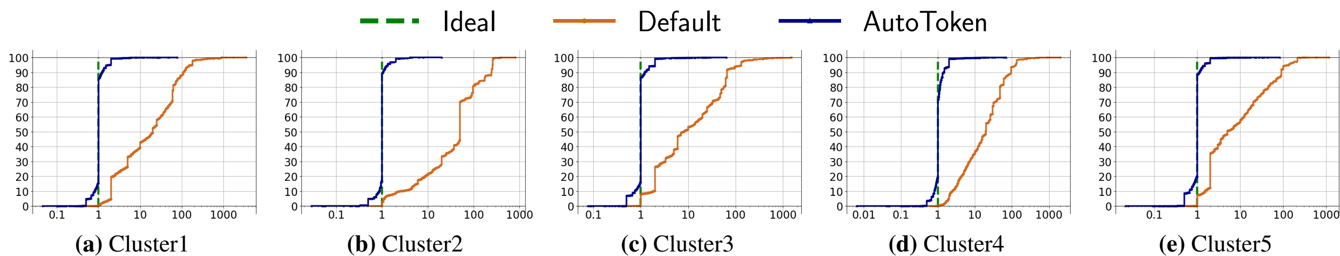
Note that DT is not available at training time and so we need to select which signatures to train models for using only information from DR. An option is to consider only those signatures in DR that have allocation ratio ≥ 1 for all jobs in DR having those signatures. Figure 10b shows the maximum coverage in DT with this selection criteria in DR and assuming perfect per-signature models that we denote as OPT. These numbers show an upper bound as OPT models incur no error whereas realistic models can lead to some signatures being discarded in the validation phase due to large errors. We show S = 1 for completeness; in practice, we use S > 1 and usually S = 10 by default as we want to have multiple jobs for training a model. Although the OPT coverage is lower than in Figure 10a, ranging from 12.9% (S = 2, Cluster2) to 30% (S = 2, Cluster5), this is a substantial portion of our workloads that can be targeted while meeting our production requirements.

In consultation with our product teams we decided to adopt the above criteria to select production candidates — job groups to train models for AutoToken. We focus our evaluations only on these production candidates in the rest of the paper.

Table 3: Number of filtered signatures (distinct) and corresponding jobs, as a percentage of total, in the DR datasets, subject to S = 2 and no under-allocated jobs. Figure 6 shows their distributions.

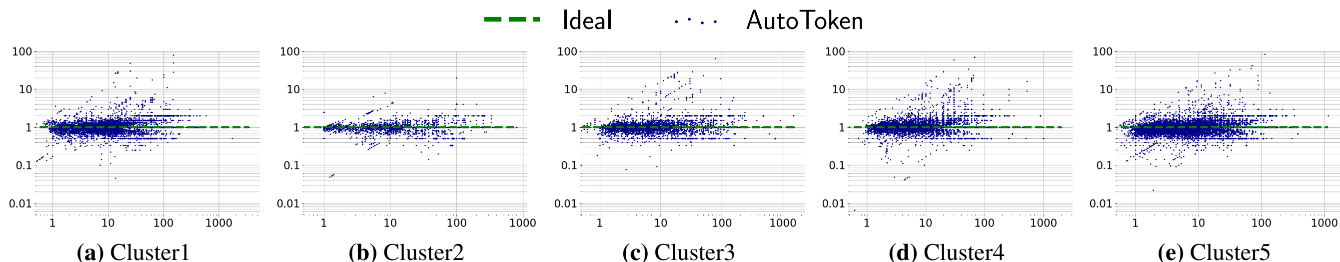
	Cluster1	Cluster2	Cluster3	Cluster4	Cluster5
#Sig1	1.7%	3.0%	1.4%	1.8%	1.1%
#Sig2	2.1%	3.2%	1.5%	2.3%	1.1%
#Sig1_Jobs	23.9%	30.6%	55.2%	35.0%	36.0%
#Sig2_Jobs	25.8%	31.6%	48.0%	35.5%	34.6%

Table 3 shows the number of groups for each signature type and jobs belonging to those groups, both as a percentage of the total number of jobs in DR, for the production candidates with S = 2. Figure 6 further shows distributions of the number of groups, and



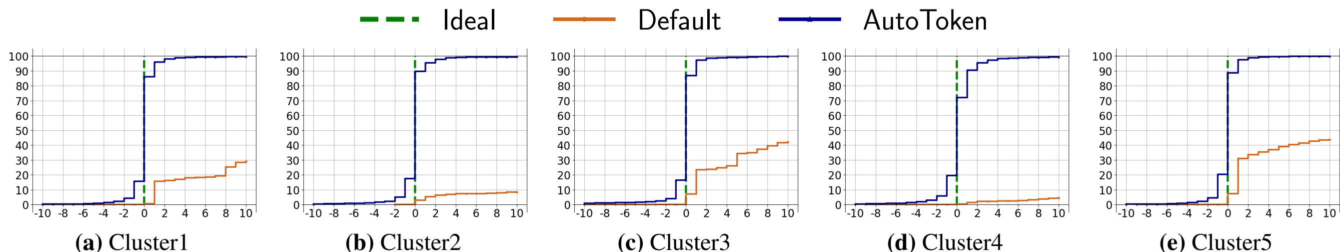
(a) Cluster1 **(b)** Cluster2 **(c)** Cluster3 **(d)** Cluster4 **(e)** Cluster5

Figure 7: Distributions of the percentage of predicted jobs (y-axis) for DT datasets as a function of the allocation ratio (= Requested/Peak tokens, x-axis, log scale) with AutoToken (S = 10, LR_NS model) and with Default allocations for those jobs.



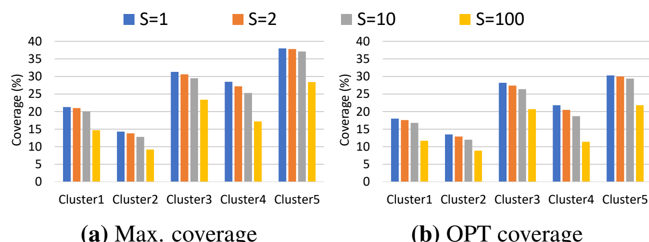
(a) Cluster1 **(b)** Cluster2 **(c)** Cluster3 **(d)** Cluster4 **(e)** Cluster5

Figure 8: Comparisons between allocation ratios using AutoToken (y-axis, log scale) and Default values (x-axis, log scale) for the above set of predicted jobs. Each dot represents a predicted job.



(a) Cluster1 **(b)** Cluster2 **(c)** Cluster3 **(d)** Cluster4 **(e)** Cluster5

Figure 9: Distributions of the percentage of predicted jobs (y-axis) as a function of (Requested − Peak) allocations (x-axis) within the interval [-10,10] using AutoToken and Default values for the above set of predicted jobs. (Requested − Peak) allocation values of >, =, and < 0 denote over-, exact-, and under-allocations respectively.



(a) Max. coverage **(b)** OPT coverage

Figure 10: Max. coverage on DT for (a) signatures appearing in DR, and (b) for OPT model trained on recurring signatures in DR with non-underallocated jobs.

jobs belonging to those groups, as a function of $n(g)$ within this subset. The distributions of the number of groups and jobs are different — for a given value of $n(g)$, there is a higher percentage of groups having less than or equal to that value compared to the percentage of jobs. Thus, there are more groups with few jobs and fewer groups with more jobs. Interestingly, within each cluster, the counts, average jobs per group, and distributions for the two grouping functions are similar to one another, with small differences.

Increasing the support requirement from S = 2 to S = 10 significantly improves accuracy (see Section 5.2) due to the availability of more data for training the model for a group. Interestingly, although this also eliminates a large fraction of groups, e.g., ~ 60% for Cluster5 (see Figure 6) from consideration for training the models, the impact on prediction coverage is small. Hence, we use S = 10 as the default configuration for training AutoToken models. S = 10 reduces the number of filtered signatures to 0.3% (Cluster1) – 0.8% (Cluster2) of total jobs in DR for both $Sig1$ and $Sig2$.

4.3 Prediction Coverage and Accuracy

For evaluating the predictions made by AutoToken, we use the Testing dataset (DT) for each cluster. Unlike the Training dataset (DR), we consider all jobs in DT, that is, we do not filter out jobs belonging to groups below the support threshold or under-allocated jobs. Indeed, in the production environment, the actual peak resource usage would not be known prior to the job execution. Table 4 shows the prediction coverage and accuracy using only $Sig1$, using both $Sig1$ and $Sig2$, and for OPT. Using both signatures increases the coverage by 0.8% (Cluster5) – 2.7% (Cluster2) from using only $Sig1$. Although the impact on overall error (RMSE) is cluster-dependent — it may increase (Cluster1), decrease (Cluster2), or stay roughly the same (Clusters 3–5) — it is < 5 except for Cluster2. Prediction accuracy for Cluster2 is improved with a shorter training interval (see Section 5.3).

The difference between $Sig1$+$Sig2$ and OPT is that, the latter has RMSE = 0 and no models are discarded in the validation step. Although validation reduces coverage by 1.1% (Cluster1) – 1.8% (Cluster3), it improves accuracy — without it, RMSE is higher for

Table 4: Prediction metrics on DT, with model training on DR, S = 10, using only $Sig1$, and using both $Sig1$ and $Sig2$.

(a) Prediction coverage = predicted/total jobs

	Cluster1	Cluster2	Cluster3	Cluster4	Cluster5
Sig1	14.2%	8.0%	23.7%	15.8%	27.3%
Sig1+Sig2	15.6%	10.7%	24.6%	17.0%	28.1%
OPT	16.8%	12%	26.4%	18.7%	29.4%

(b) Prediction RMSE (Root Mean Square Error)

	Cluster1	Cluster2	Cluster3	Cluster4	Cluster5
Sig1	3.5	16.0	4.6	4.5	2.2
Sig1+Sig2	4.8	14.2	4.5	4.6	2.3
OPT	0	0	0	0	0

all clusters, up to 33.2 for Cluster1 and 22.2 for Cluster2.

Cumulative distributions. Figure 7 shows the distributions of percentage of predicted jobs as a function of the (Requested/Peak) ratio for the same DT datasets. Overall, we see that AutoToken predictions are much closer to the Ideal compared to Default. There are still over-allocations for predicted jobs — 28% for Cluster4, 10.3% – 13.9% for other clusters. In contrast, 92.6% (Cluster5) – 99.7% (Cluster4) of these jobs are over-allocated with the Default allocation. However, 15.8% (Cluster1) – 20.3% (Cluster5) of predictions cause under-allocations, as opposed to ≤ 0.2% with the Default allocations for AutoToken-predicted jobs. The worst-case under-allocation is also larger than with Default allocations, but as we discuss below, the vast majority of absolute errors are small.

Error spread. Although Figure 7 shows distributions of allocation ratios with AutoToken and Default for the same set of jobs, it does not show how the ratios relate for individual jobs. To gain insight into this, we consider the error spread shown in Figure 8. Each point in the scatter plot corresponds to a single AutoToken-predicted job with the allocation ratios for Default and AutoToken on the x- and y-axes respectively. We observe that the vertical spread is much smaller than the horizontal spread, indicating huge improvement with AutoToken in allocation ratios for individual jobs. For example, for Cluster3, 74.5% and 99.2% of points lie within the y-axis intervals of [0.9, 1.1] and [0.5, 2] respectively, but the corresponding x-axis intervals for those jobs are [0.45, 1500] and [0.42, 1500] respectively. Ideally, with perfect AutoToken predictions, all points should be aggregated along the (0,1) line.

Absolute error. Although there are some errors in the AutoToken predictions, the absolute difference can be small if the actual peak itself is small. This happens for quite a few jobs and such errors can be tolerated in practice. Figure 9 shows the distribution of the error (difference from ideal value), zooming in on the interval [-10, 10]. For all clusters, this error range comprises ≥ 99% of AutoToken predictions, with 92.5% (Cluster4) – 96.7% (Cluster5) within the error range [-2, 2] and 84.6% (Cluster4) – 93.1% (Clusters 3, 5) within [-1, 1]. Thus, the vast majority of AutoToken prediction errors are within 2 tokens or less for these datasets. Errors with Default token allocations are considerably larger, for example, 56.6% (Cluster5) – 95.7% (Cluster4) of the predicted jobs have over-allocation of 10 tokens or more with the Default allocations.

5. DESIGN CHOICES

So far we have discussed prediction results for a single, default, configuration (LR_NS, S = 10). We will now discuss the rationale for those settings and other choices for AutoToken's design.

5.1 Choice of Model

AutoToken uses LR_NS (Linear Regression models with inputs Not Standardized) model by default. To explore the impact on model accuracy and coverage with other models, we explore the following models — Linear Regression (LR), AdaBoost Regression (ABR), Gradient Boosting Regression (GBR), and Random Forest Regression (RFR). Except for LR_NS, we standardize the inputs for all other models. We use a fixed random seed and 100 estimators each for ABR, GBR, and RFR.

Figure 11 shows boxplots for the prediction accuracy, on the DT datasets, with different models. Ideally, the ends of the boxes (quartiles) and whiskers would all be at 1 and there would no outliers. We observe that for all clusters, regardless of the choice of model, the quartiles are very close to 1. The whiskers are also almost identically placed across models, showing hardly any impact in reducing the [1, 99]th percentile spread of values of the allocation ratio. However, the outliers vary depending on the model indicating that there can be a difference in the worst-case predictions depending on the model. Usually, the more sophisticated models — ABR, GBR, RFR (except Cluster3) outperform the linear models for the worst-case. The simpler linear models are much faster to train and use for predictions, so we use them by default for AutoToken. Further, there is hardly any difference between LR_NS and LR indicating little benefit of standardization. We choose to use the non-standardized version by default as that does not require having the standardization constants available for prediction and also simplifies the computation for prediction.

5.2 Model Support

AutoToken uses S = 10 by default. Figure 12 shows the coverage and errors (RMSE), on the DT datasets, for the predicted token counts using different values for the support, S, and for different models. We include OPT as a reference for the maximum coverage and minimum error achievable in our framework.

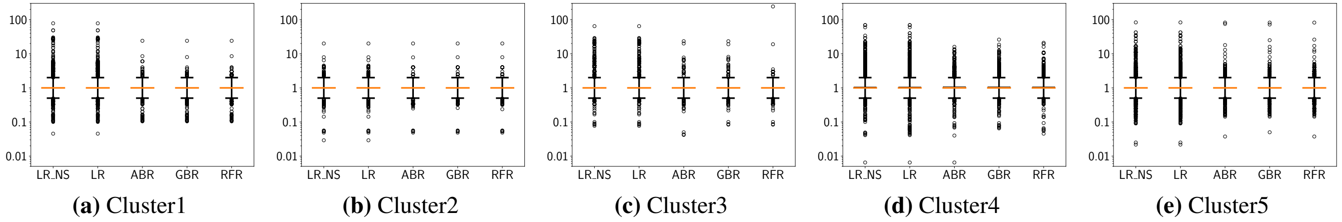
Training with few data points risks over-fitting models, resulting in large prediction errors on the testing dataset. We observe that the RMSE reduces with increasing values of S, with a significant RMSE reduction in increasing S from 2 to 10 for Cluster2 that has the smallest number of jobs in DR (and DT) among all clusters. On the other hand, while increasing S reduces coverage due to fewer models being trained, the reduction is much more pronounced for S = 100 vs S = 10 than for S = 10 vs S = 2 since signatures appearing frequently in the past (DR) are more likely to appear again in future (DT). Overall, S = 10 provides a good operating point with a large accuracy improvement and low coverage loss.

For a given cluster and value of support, both coverage and errors depend on the model. Coverage is affected by the validation step that discards models based on its accuracy and the validation criteria. There is a slight increase in coverage beyond LR_NS with more sophisticated models such as RFR, e.g., 28.2% to 28.8% for Cluster5, S = 10. Similarly, the accuracy improvement, is small, e.g., RMSE of 2.26 to 2.05 for Cluster5, S = 10. Cluster1 sees the most improvement — RMSE of 4.79 to 1.83 for S = 10, but the absolute RMSE values are still small (except for Cluster2, that as we discuss in Section 5.3, is affected by training interval length).

Overall, we conclude that the value of support, S, affects coverage and accuracy more than the choice of the model. Simple models perform well and there is little incentive to use more sophisticated, computationally-expensive models.

5.3 Training Interval

Figure 13 The prediction coverage and accuracy also depend on the length of the training interval. We study this effect by using subsets of dataset DR for the training while using the same dataset DT for testing. Figure 13a shows the different intervals. L4, L3, L2, L1 are 1, 2, 3, are 4 weeks long respectively and L0 covers



(a) Cluster1 **(b)** Cluster2 **(c)** Cluster3 **(d)** Cluster4 **(e)** Cluster5

Figure 11: Boxplots for allocation ratio (= Predicted/Peak tokens) distributions for predicted jobs using different models with S = 10. The y-axis (log scale) shows the allocation ratio, with each whisker encompassing the [1, 99]th percentile range of ratios for that model. The prediction coverage varies slightly with the model (see Figure 12 for S = 10) due to the validation step.

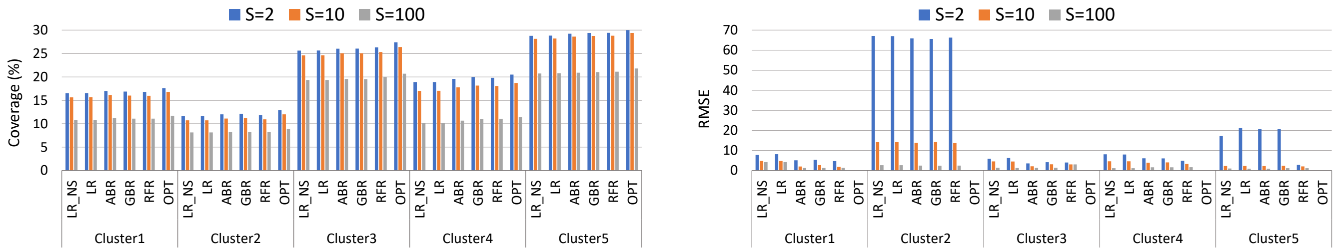
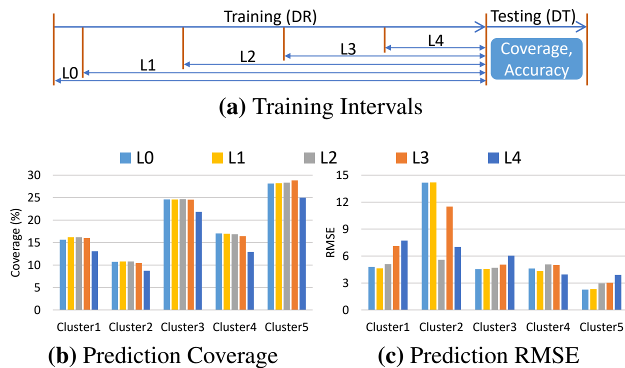


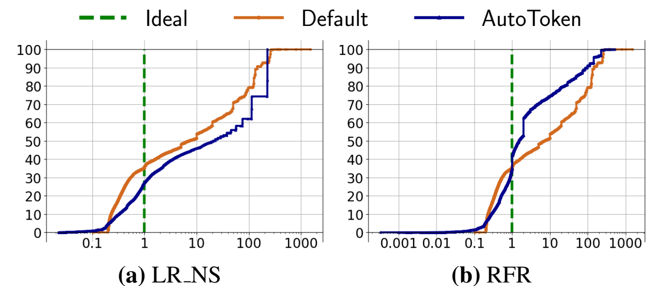
Figure 12: Prediction coverage and RMSE for predicted jobs using different models and values for the support S.



(a) Training Intervals

(b) Prediction Coverage **(c)** Prediction RMSE

Figure 13: **(a)** Different training intervals depending upon the length of past history and corresponding prediction **(b)** coverage and **(c)** accuracy (RMSE) for the testing dataset (DT).



(a) LR_NS **(b)** RFR

Figure 14: Distributions of the percentage of predicted jobs (y-axis) on Cluster2 as a function of the allocation ratio (= Requested/Peak tokens, x-axis, log scale) with **(a)** LR_NS and **(b)** RFR models for AutoToken global predictor, and Default allocations for those jobs.

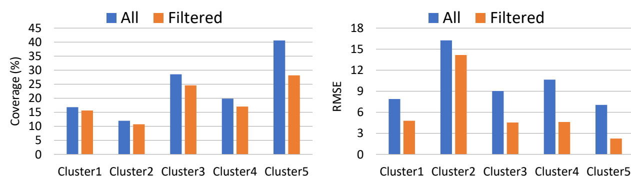
the whole month. Figures 13b and 13c show the coverage and accuracy with default AutoToken configurations (LR_NS, S = 10). For all clusters, increasing training length from 1 (L4) to 2 weeks (L3) improves coverage since S = 10 precludes training for daily-recurring jobs for L4. It also improves accuracy (lower RMSE) for clusters other than Clusters 2 and 4. While L2 coverage is almost similar to L3, it tends to improve accuracy, e.g., for Cluster1, Cluster2. Interestingly, however, longer training intervals may not always help in improving accuracy, e.g., L1, L0 for Cluster2, due to the changing nature of the workloads over time. Thus, we conclude that, for these datasets, a training interval over the past 3 weeks (L2) provides good prediction coverage and accuracy.

5.4 Global Predictor

AutoToken trains and uses per-signature models. Another option could be to train a predictor over all jobs, which we call a global predictor. It uses the same features as the per-signature predictor, but does not group jobs separately based on their signatures. Such a global predictor would be able to predict token counts for any new job as it would not be restricted to signature matches. Thus, prediction coverage would be 100%. However, we observed that the prediction accuracy is low. Figure 14 shows distributions of jobs as a function of the allocation ratio (= Requested/peak tokens) for the testing dataset (DT) of Cluster2. The distributions for AutoToken are far from Ideal, and the RMSE was 634.7 with LR_NS model and 550.3 with RFR which is substantially higher than the RMSE of 14.2 with the per-signature models! Apart from building highly accurate specialized models, the signature-based approach also allows discarding specific signatures based on some criteria, e.g., under-allocated jobs, poor validation (fitting) results, etc.

5.5 Expanding Production Candidates

AutoToken trains models only for signatures such that no job belonging to that corresponding group is under-allocated. One could relax this restriction and train for all signatures, but still have the same validation step for keeping or discarding models. Figure 15 shows prediction coverage and accuracy for this policy ('All') along with those of the default ('Filtered') policy. 'All' can significantly improve coverage, e.g., 40.6% instead of 28.2% for Cluster5, but comes with a cost of reduced accuracy, e.g., RMSE of 7.1 instead of 2.3 for Cluster5. Thus, we use the 'Filtered' policy by default, not



(a) Prediction Coverage **(b)** Prediction RMSE

Figure 15: (a) Prediction coverage (%) and **(b)** accuracy (RMSE) for the testing dataset (DT) with all-groups and filtered (default) configuration for AutoToken.

just for better accuracy but also due to our production requirement to not stress the clusters with more resource requests (**R5**).

6. BASELINE COMPARISONS

In this section, we first compare AutoToken with two strong baselines, and then we discuss other related work.

6.1 Conservative Allocation

Given that our focus is to avoid over-allocation, we first consider the most conservative policy that predicts the peak resource usage of a job as the maximum seen, in the past, over all jobs belonging to the same group. We call this strategy *MaxPeak*. MaxPeak uses the normalized job name as a grouping function, trains over all groups, and predicts a single peak value for all jobs in the group. MaxPeak has very little computational requirements and therefore it is the fastest in terms of both training and scoring.

As before, we train AutoToken on production candidates and compare with MaxPeak over the same sets of jobs. Table 5 and Figure 16 compare the prediction results. Both AutoToken and MaxPeak have lower RMSE than Default, significantly reducing the over-allocations and having closer-to-Ideal distributions. However, they do have some under-allocations: 10.6% (Cluster2) – 22.3% (Cluster4) for AutoToken, 0.5% (Cluster5) – 1% (Clusters 1 and 4) for MaxPeak, vs 0.1% for Default. As we have discussed earlier 4.3, most under-predictions for AutoToken have small absolute values that are easily tolerable with spare tokens. Compared to AutoToken, MaxPeak results in a lower percentage of under-allocations, but not necessarily in the worst-case, e.g., -101 tokens vs -3 for AutoToken for Cluster2.

6.2 Best-fit Allocation

Morpheus [23] computes a job resource model given the resource usage skylines of past occurrences of a recurring job. It computes a new skyline that best fits the set of skylines for the job executions subject to constraints on the extent of over-allocation and under-allocation of resources. The extents are determined from values of two hyper-parameters.

AutoToken differs from Morpheus in four ways. First, Morpheus groups jobs by job names after normalizing instance-specific fields such as job ids and timestamps. In contrast, AutoToken groups jobs by plan-based signatures, which are more accurate in identifying recurring jobs and avoid grouping unrelated jobs with similar names. We looked at jobs over a day's interval for each of our clusters and found that, depending on the cluster, 12% – 31.3% job groups (by normalized name) have > 1 distinct primary signature ($Sig1$) values for jobs within the group, 2.3% – 9.5% have ≥ 10, and 0.1% – 0.5% have ≥ 100 distinct values for $Sig1$.

Second, Morpheus generates a single skyline for a group, with the same peak for all the jobs in that group. As a result, it misses

Table 5: RMSE for the common set of predicted jobs.

	Cluster1	Cluster2	Cluster3	Cluster4	Cluster5
AutoToken	2.8	3.1	1.3	3.5	8.3
MaxPeak	76.6	115.8	24.6	57.6	18.0
Morpheus	164.0	138.2	66.6	384.7	198.2
Default	305.2	185.4	111.6	82.9	76.9

fluctuations in the input data, job parameters, or even plan structure, within the group. For the single day's test data mentioned above, we found that depending on the cluster, 16.4% – 24.6% of groups have a Coefficient of Variation (CV) ≥ 10%, and 0.3% – 3.9% have CV ≥ 100%, in intra-group per-job peak tokens used. Consequently, using a single peak value for all jobs in the group would result in errors. In contrast, AutoToken treats these variables as features and adjusts the peak for each job instance.

Third, Morpheus models the problem as linear programs which are much more computationally intensive, often taking hours with off the shelf open source solvers. In contrast, linear regressions in AutoToken terminate way quickly. Finally, Morpheus needs the resource-usage skylines from past executions as input whereas AutoToken only needs the peak and a few feature values. Saving skylines for long-running jobs may incur overheads in storage costs.

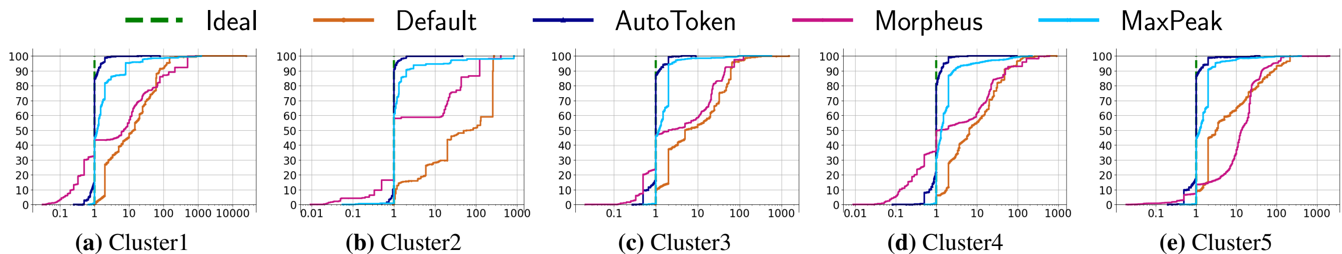
For each job, we extracted its resource skyline with 10-second granularity and fed it to the single threaded Morpheus implementation on a single machine. We use the LP solver from ojAlgo [4] for Morpheus training with a 1-hour timeout period per group. Similar to the AutoToken's default configuration, we use S = 10, i.e., only consider groups with $n(g)$ ≥ 10. Due to the long model-training times for Morpheus, we study its coverage and accuracy using a smaller training and testing dataset, each of which correspond to one day's set of jobs on that cluster with the testing interval succeeding the training interval. Note that one could potentially improve Morpheus performance using commercial LP solvers but that was beyond the scope of this work. Table 5 and Figure 16 show that compared to Default, Morpheus has lower RMSE except for Clusters 4 and 5, but can result in a large under-allocations, e.g., 32.6% (Cluster1), 36.2% (Cluster4). Overall, MaxPeak is more accurate than Morpheus, but AutoToken outperforms them both with up to two orders of magnitude lower RMSE and a smaller error spread.

6.3 Other Related Work

We now present a qualitative discussion on other related work in resource optimization. Other than Morpheus, Jockey [15] was also proposed in the context of SCOPE and it uses an offline simulator to mimic the behavior, with some simplifications, of the SCOPE job scheduler and estimate the runtimes of recurring SCOPE jobs. The simulator inputs include performance statistics and per-stage timing information from past job executions, along with job operators, stages, dependencies, and a target token count. AutoToken uses simple, closed-form models that do not need information about job timing or scheduling and hence can be more easily deployed and evaluated at scale, without requiring expensive simulation.

Ernest [37] predicts the performance of analytics jobs using a black-box approach. It involves running instances of the job on smaller scales of input data and cluster size. Thereafter, Ernest uses the performance data to determine the parameters of a model. In contrast, AutoToken uses job characteristics that are known/estimated before dispatch to determine model parameters and does not need to profile job performance with a different number of tokens.

Perforator [31] uses a gray-box approach for estimating performance impact of resource allocations for BigData queries. This involves using semantic analysis, sampling, profiling, and non-linear regression for data-size estimation; black-box profiling with cali-



(a) Cluster1 **(b)** Cluster2 **(c)** Cluster3 **(d)** Cluster4 **(e)** Cluster5

Figure 16: Distributions of the percentage of predicted jobs (y-axis) as a function of the allocation ratio (= Requested/Peak tokens) (x-axis, log scale) with AutoToken, Morpheus, MaxPeak, and Default allocations for those jobs. Allocation ratios of >, =, and < 1 denote over-, exact-, and under-allocations respectively. These are a common set of jobs that have predictions for all the different techniques.

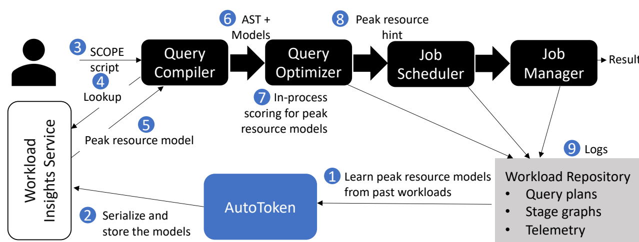
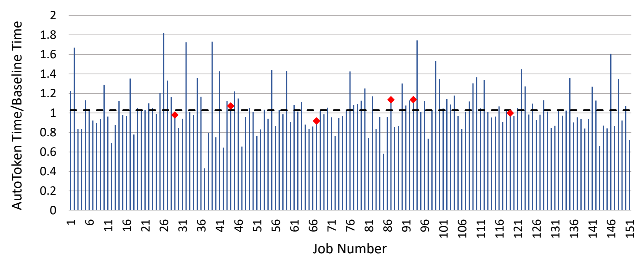


Figure 17: Putting it all together.

(a) AutoToken predictions (blue bars) normalized to Default allocations (orange bars).



(b) Performance impact due to AutoToken-predicted allocations.

Figure 18: **(a)** Token allocations, with AutoToken (darker foreground) normalized to Default (lighter background) and **(b)** slowdowns, for AutoToken-predicted jobs in a customer VC. Baseline refers to job flighting with the Default token count. The red diamonds (◆) mark the six jobs that AutoToken under-predicted with respect to the peak allocations for those jobs. The dashed line in **(b)**, at ~1.027, corresponds to the ratio of the sum of AutoToken times to the sum of Baseline times for all predicted jobs.

bration queries for hardware performance modeling; white-box analytical models for framework performance modeling. In contrast, AutoToken does not need profiling for data size estimation — it reuses metadata information about input streams and estimates generated by the SCOPE query optimizer. Also, its prediction models do not assume anything about the structure of the job execution framework or hardware characteristics.

There is large body of prior work on Hadoop MapReduce, including learning cost models for MapReduce [35], cluster sizing and self tuning [17, 18], applying resource constraints [20], workload modeling [40], job performance model [25], building resource profile [38], optimal resource provisioning [29], and for graph processing [10]. However, the complexity and structure of query DAGs have evolved since the early days of Hadoop MapReduce. Recent works propose simulation based approaches for performance prediction in Spark [7, 12]. AutoToken, in contrast, leverages actual production workloads and recurring job patterns to learn the peak resource model. Others have considered forecasting workload at the machine level [24, 19, 36, 26]. AutoToken leverages the recurring job patterns to avoid the forecasting problem and also considers workload at the query level. Still others have considered dynamic and global nature of the resource allocation problem [6, 28, 8, 27, 5, 13, 14]. Overall, AutoToken focuses on peak resource usage prediction, but not performance prediction of jobs. This suffices as the goal is to avoid over-allocation (with no performance impact) and we are able to achieve this with simple models, albeit with a small (tolerable) fraction of under-allocated job predictions.

7. PUTTING IT ALL TOGETHER

We have implemented AutoToken, as part of the broader workload optimization platform Peregrine [21], as per requirement **R7**, and integrated it with the SCOPE query engine. Figure 17 illustrates the end to end system integration. Starting from the workload repository that consists of SCOPE query plans, stage graphs, and associated telemetry seen in the past, AutoToken learns the peak resource models (Step 1) and stores them into a workload insight service (Step 2). We focus on over-allocated jobs since that will reduce the resource consumption without affecting the runtime performance, i.e., a win-win for both the customers and the service operators. Therefore, we filter out the under-allocated jobs and load models only for the over-allocated jobs into the workload insight service. For a new SCOPE job, users can specify an additional flag for enabling AutoToken (Step 3), as per requirement **R8**. Once AutoToken is enabled, the SCOPE compiler loads the peak resource models from the workload insight service (Step 4 and 5), and passes the models along with the query AST to the query optimizer (Step 6). The query optimizer provides the features to score the peak resource model (Step 7) and passes the peak resource hint (the token count) to the job scheduler (Step 8). The job scheduler takes care of enforcing the peak resource when scheduling the job for execution. All query processing logs are finally collected back into the workload repository (Step 9), thereby completing the end-to-end feedback loop. The training overhead is in the range of a few hours for a large cluster (over hundred thousand jobs) while the feedback loop latencies are in the range of few milliseconds.

Figure 18 shows the token savings and performance impact for AutoToken-predicted jobs on a customer VC. The training interval for this experiment was for three weeks and the testing interval was for the subsequent three days. AutoToken predicted peak token counts for 151 jobs out of a total of 692 jobs (21.8% coverage) for this VC in the testing interval. Figure 18a shows, for each of the 151 jobs, the predicted token counts (foreground, darker color) normalized to the Default values (background, lighter color). For these jobs, the median token savings was 97.5% and the sum of tokens predicted was 7.6% that of the sum of Default token counts for those jobs. To understand the performance impact, we reran the production jobs with the same inputs, once for the Default allocation (baseline) and once with the predicted token counts, similar to in prior works [39, 34]. For six jobs, specially marked in Figure 18b, the predicted tokens were below the actual peak (under-allocation), but their performances were close to that of the baseline. Although we ran each job for 8–20 times for both the baseline and predicted tokens, we observed some variance in the average run times — an issue that has been discussed in prior work [33]. As a result, while the sum of the average run times with the predicted token counts was 2.7% higher than with Default counts, this is mostly due to the runtime variances in the cloud. We conclude that there are substantial token savings with AutoToken without a noticeable loss of performance in these jobs.

8. REPEATABILITY

To encourage further research on resource estimation for our workloads, we will release[1] a dataset simulation tool that can generate datasets of arbitrary size using characteristics from the real datasets that we have used. The simulator takes as inputs distribution information for multiple recurring jobs and can generate arbitrary number of instances per job. We have extracted the mean and standard deviation of each of the columns, and the covariance matrix across of all the columns. The simulator separates the independent columns, i.e., columns having no correlation with any other column. Different sets of columns can correlate for different job groups, so it is not possible to discard all such columns statically from the entire dataset. For correlated columns, we compute the Cholesky decomposition [2] of the truncated covariance matrix, generate normal distribution dataset of desired size (as specified by the user), and then shape the normal distribution using the desired mean and factorization. The simulated dataset Y is given as:

$$Y = M + A \cdot Z \tag{1}$$

where M is the mean vector, A is the Cholesky decomposition, and Z is the normal distribution of desired size. For independent columns, we simply shape the normal distribution using the mean and standard deviation, i.e., Y for independent columns is given as:

$$Y = M + SD \cdot Z \tag{2}$$

where SD is the standard deviation vector.

Divergence. To verify that the simulated datasets are close to the actual ones, we computed the Kullback-Leibler divergence [3] between the actual and simulated datasets (having k dimensions each), with mean vector and covariance matrix M_0, M_1, C_0, C_1, respectively, as follows:

$$D_{\mathrm{KL}}(d_0 \| d_1) = \frac{1}{2}\Big(\mathrm{tr}(C_1^{-1}C_0) + (M_1 - M_0)^T C_1^{-1}(M_1 - M_0) - k + \ln\left(\frac{\det C_1}{\det C_0}\right)\Big) \tag{3}$$

[1] https://github.com/microsoft/Peregrine

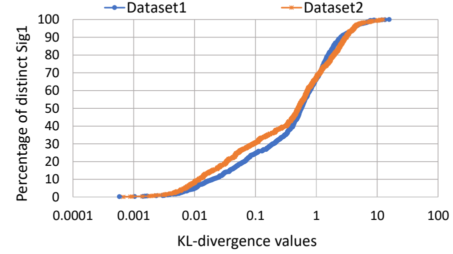
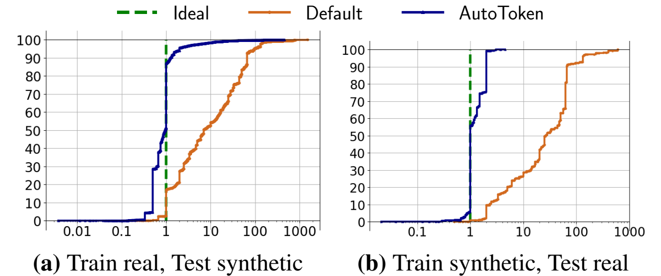


Figure 19: Distributions of job groups ($Sig1$) as a function of KL-divergence values (x-axis, log scale) for Dataset1 and Dataset2.



(a) Train real, Test synthetic **(b)** Train synthetic, Test real

Figure 20: AutoToken evaluation results using scaled datasets generated. In both figures, the x-axis in log scale shows the ratio of (Requested/Peak) allocations and y-axis shows the percentage of predicted jobs. **(a)** shows the distributions for models trained on the real dataset (DR) and tested on the generated dataset (Dataset2). **(b)** shows distributions for models trained on the generated dataset (Dataset1) and tested on the real dataset (DT).

We generated two datasets from Cluster3, S = 10: Dataset1 from DR, and Dataset2 from DT. Dataset1 has 995K jobs while Dataset2 has 990K jobs. Figure 19 shows the distributions of distinct $Sig1$ values as a function of KL-divergence values for these datasets. Smaller KL-divergence values indicate a better fit between the simulated and actual datasets. The user may choose to retain a subset of the generated datasets based on the KL-divergence values.

Validation. We used our models on the generated datasets to see if similar prediction profiles are obtained. Figure 20 shows the distributions of (requested/peak) token allocation ratios for two scenarios — (a) models trained on DR, Cluster3 and tested on Dataset2, and (b) models trained on Dataset1 and tested on DT, Cluster3. Both results show improvements by AutoToken over default allocations, similar to those observed with real datasets. The synthetic datasets also show the deviation of the AutoToken predictions from the Ideal values, again similar to those observed with real datasets.

9. CONCLUSIONS AND FUTURE WORK

In this paper we study the problem of estimating peak resource requirements of jobs to reduce over-allocations and describe AutoToken, a simple but effective predictor that we have developed for this task. We show how AutoToken improves upon state-of-the-art approaches for resource prediction by using signatures to identify recurring jobs, and linear models that learn peak token counts using features such as job costs, cardinalities, input sizes, etc. from past executions of recurring jobs. We also describe AutoToken's prediction pipeline and discuss how it is integrated into the Peregrine workload optimization infrastructure at Microsoft. Further improvements to both prediction coverage and accuracy may be possible by considering the shape of the job plan as an additional input feature to the models. We plan to explore this in future work.

10. REFERENCES

[1] Amazon Athena. https://aws.amazon.com/athena/.

[2] Cholesky decomposition. https://en.wikipedia.org/wiki/Cholesky_decomposition.

[3] Kullback-Leibler divergence. https://en.wikipedia.org/wiki/Kullback-Leibler_divergence.

[4] oj! Algorithms. https://www.ojalgo.org/.

[5] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 469–482, USA, Mar. 2017. USENIX Association.

[6] M. Amiri and L. Mohammad-Khanli. Survey on prediction models of applications for resources provisioning in cloud. *Journal of Network and Computer Applications*, 82(C):93–113, Mar. 2017.

[7] D. Ardagna, E. Barbierato, A. Evangelinou, E. Gianniti, M. Gribaudo, T. B. M. Pinto, A. Guimarães, A. P. Couto da Silva, and J. M. Almeida. Performance prediction of cloud-based big data applications. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, IKPE '18, pages 192–199, New York, NY, USA, 2018. Association for Computing Machinery.

[8] A. Biswas, S. Majumdar, B. Nandy, and A. El-Haraki. Automatic resource provisioning: A machine learning based proactive approach. In *Proceedings of the 2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, CLOUDCOM '14, pages 168–173, USA, Dec. 2014. IEEE Computer Society.

[9] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 285–300, USA, 2014. USENIX Association.

[10] A. Cela, Y. C. Lee, and S.-H. Hong. Resource provisioning for memory intensive graph processing. In *Proceedings of the Australasian Computer Science Week Multiconference*, ASCW '18, New York, NY, USA, 2018. Association for Computing Machinery.

[11] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[12] Y. Chen, J. Lu, C. Chen, M. Hoque, and S. Tarkoma. Cost-effective resource provisioning for Spark workloads. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, pages 2477–2480, New York, NY, USA, 2019. Association for Computing Machinery.

[13] A. Chung, J. W. Park, and G. R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 121–134, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Z. Fan, R. Sen, P. Koutris, and A. Albarghouthi. Automated tuning of query degree of parallelism via machine learning. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[15] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. Association for Computing Machinery.

[16] Google BigQuery. https://cloud.google.com/bigquery.

[17] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[18] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Fifth Biennial Conference on Innovative Data Systems Research*, pages 261–272. www.cidrdb.org, 2011.

[19] R. Hu, J. Jiang, G. Liu, and L. Wang. KSwSVR: A new load forecasting method for efficient resources provisioning in cloud. In *Proceedings of the 2013 IEEE International Conference on Services Computing*, SCC '13, pages 120–127, USA, June 2013. IEEE Computer Society.

[20] V. Jalaparti, H. Ballani, T. Karagiannis, A. Rowstron, and P. Costa. Bazaar: Enabling predictable performance in datacenters. Technical Report MSR-TR-2012-38, Feb. 2012.

[21] A. Jindal, H. Patel, A. Roy, S. Qiao, Z. Yin, R. Sen, and S. Krishnan. Peregrine: Workload optimization for cloud query engines. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 416–427, New York, NY, USA, 2019. Association for Computing Machinery.

[22] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 191–203, New York, NY, USA, 2018. Association for Computing Machinery.

[23] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated SLOs for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 117–134, USA, 2016. USENIX Association.

[24] A. Khan, X. Yan, S. Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium*, pages 1287–1294, Apr. 2012.

[25] M. Khan, Y. Jin, M. Li, Y. Xiang, and C. Jiang. Hadoop performance modeling for job estimation and resource provisioning. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):441–454, Feb. 2016.

[26] I. K. Kim, W. Wang, Y. Qi, and M. Humphrey. Empirical evaluation of workload forecasting techniques for predictive cloud resource scaling. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 1–10, June 2016.

[27] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in*

Networks, HotNets '16, pages 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[28] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis. Efficient resource provisioning in compute clouds via VM multiplexing. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 11–20, New York, NY, USA, 2010. Association for Computing Machinery.

[29] P. P. Nghiem and S. M. Figueira. Towards efficient resource provisioning in MapReduce. *Journal of Parallel and Distributed Computing*, 95(C):29–41, Sept. 2016.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, Nov. 2011.

[31] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan. PerfOrator: Eloquent performance models for resource optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 415–427, New York, NY, USA, 2016. Association for Computing Machinery.

[32] A. Roy, A. Jindal, H. Patel, A. Gosalia, S. Krishnan, and C. Curino. SparkCruise: Handsfree computation reuse in Spark. *PVLDB*, 12(12):1850–1853, 2019.

[33] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *PVLDB*, 3(1–2):460–471, 2010.

[34] T. Siddiqui, A. Jindal, S. Qiao, H. Patel, and W. Le. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 99–113, New York, NY, USA, 2020. Association for Computing Machinery.

[35] F. Tian and K. Chen. Towards optimal resource provisioning for running MapReduce programs in public clouds. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 155–162, July 2011.

[36] C. Vazquez, R. Krishnan, and E. John. Time series forecasting of cloud data center workloads for dynamic resource provisioning. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 6(3):87–110, Sept. 2015.

[37] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 363–378, Santa Clara, CA, USA, Mar. 2016. USENIX Association.

[38] A. Verma, L. Cherkasova, and R. H. Campbell. Resource provisioning framework for MapReduce jobs with performance goals. In *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '11, pages 165–186, Berlin, Heidelberg, 2011. Springer-Verlag.

[39] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a Learning Optimizer for Shared Clouds. *PVLDB*, 12(3):210–222, 2018.

[40] H. Yang, Z. Luan, W. Li, and D. Qian. MapReduce workload modeling with statistical approach. *Journal of Grid Computing*, 10(2):279–310, June 2012.